6.3 TYPES

Julia's type system will remind you more of Python's and Ruby's than JavaScript's: there's no distinction between primitive and reference types, types are themselves objects (so there's a type called `Type`), types are organized into a hierarchy (though single-inheritance only), there are separate types for integers and floats, and you can define your own types. Julia's basic types go a bit further, distinguishing signed and unsigned integers, providing types for numbers of various bit lengths, and providing a rational number type.[1]

We show a part of the hierarchy of Julia's built-in types in Figure 6.1. Julia adds several typing concepts we've not yet seen: **abstract types**[2], **parametric types**, and **union types**. These are new and interesting concepts for us, so we'll look at each in detail.

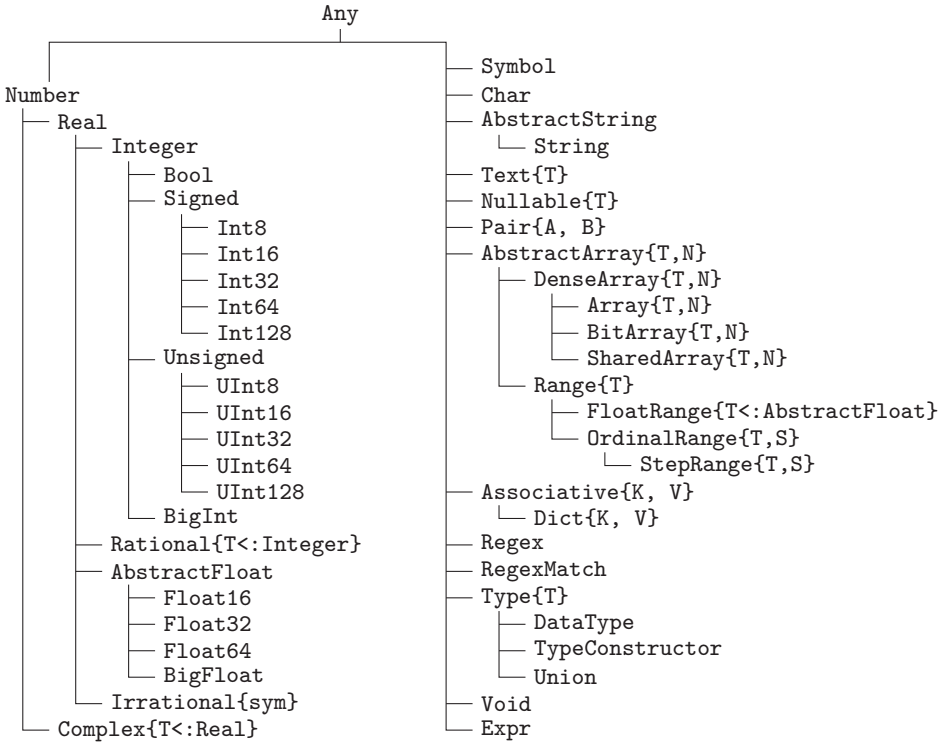

Figure 6.1 A few of the built-in Julia types

6.3.1 Abstract Types

Every type in Julia is either **abstract** or **concrete**. Every value has a single concrete type; for example, `2.54` has the type `Float64`. Let's explore a few more:

[1] You can find details of these various numeric types in Appendix A.

[2] Python does "support" the notion of an abstract class through operations in its `abc` library module, but Julia offers support for the concept directly in the language itself.

```
@assert typeof(3) == Int64
@assert typeof(0x22) == UInt8
@assert typeof(0xFA31) == UInt16
@assert typeof(false) == Bool
@assert typeof('\u263a') == Char
@assert typeof("Hello") == String
@assert typeof(BigInt(72)^3897) == BigInt
@assert typeof(:hello) == Symbol
@assert typeof(r"\d+(\.\d+)") == Regex
@assert typeof(Float64) == DataType
@assert typeof(DataType) == DataType
```

Abstract types do nothing more than *generalize* other types; for example, the abstract type `AbstractFloat` generalizes the four concrete types `Float16`, `Float32`, `Float64`, and `BigFloat`. Calling `typeof` will never produce an abstract type, but you can use `isa` to check "membership" in an abstract type:

```
@assert typeof(88) == Int64
for t in [Int64, Integer, Signed, Real, Number, Any]
  @assert isa(88, t)
end
```

In Julia, *only* abstract types can have subtypes; in fact, in Figure 6.1, all leaf types are concrete and all non-leaf types are abstract.[3] We can explore the type system with the built-in methods `super` and `subtypes` (which do what you'd expect), and the `<:` operator, which determines whether one type is a descendant of another.

```
@assert supertype(Int32) == Signed
@assert supertype(Signed) == Integer
@assert supertype(Integer) == Real
@assert supertype(Symbol) == Any
@assert Set(subtypes(Type)) == Set([DataType, TypeConstructor, Union])

@assert Float64 <: Real
@assert isa(subtypes(Type), Array)
@assert isa(Array, Type)
```

Let's implement our own abstract and concrete types. We'll do so by directly translating our little animals script from Python and Ruby. We create an abstract type `Animal`, and three concrete subtypes. As before, all animals `speak` the same way, but each makes its own `sound`:

```
abstract Animal
speak(a::Animal) = "$(a.name) says $(sound(a))"

type Horse <: Animal
  name
end
sound(h::Horse) = "neigh"
```

[3] This is *not* the case in every language. Plenty of other languages allow concrete types to have subtypes.

```
type Cow <: Animal
  name
end
sound(c::Cow) = "moooo"

type Sheep <: Animal
  name
end
sound(s::Sheep) = "baaaa"

s = Horse("CJ")
@assert speak(s) == "CJ says neigh"
c = Cow("Bessie")
@assert speak(c) == "Bessie says moooo"
@assert speak(Sheep("Little Lamb")) == "Little Lamb says baaaa"
```

The Julia version looked much different than before! We did not create classes, nor did we embed `speak` and `sound` inside the type declarations. We did, however, write three distinct `sound` implementations. When the expression `sound(a)` is executed, Julia will **dispatch** to the correct implementation *based on the type of* `a`. In Julia's terminology, `sound` is a **generic function** with three separate **methods**. The function `methods` reports the methods for a generic function, as we show here in the REPL:

```
julia> sound
sound (generic function with 3 methods)

julia> methods(sound)
# 3 methods for generic function "sound":
sound(h::Horse)
sound(c::Cow)
sound(s::Sheep)
```

6.3.2 Parametric Types

Julia types can be parameterized by other types and certain other values. We can get a feel for these types by studying a few examples:

```
@assert typeof(1:10) == UnitRange{Int64}
@assert typeof(0x5 // 0x22) == Rational{UInt8}
@assert typeof(5 // 34) == Rational{Int64}
@assert typeof(8.75im) == Complex{Float64}
@assert typeof(e) == Irrational{:e}
@assert typeof([5,3]) == Array{Int64,1}
@assert typeof([3, "abc"]) == Array{Any, 1}
@assert typeof([]) == Array{Any, 1}
@assert typeof([1 0; 0 1]) == Array{Int64, 2}
@assert typeof(Set(4)) == Set{Int64}
@assert typeof(Set(['3', '$'])) == Set{Char}
```

To distinguish sets of integers from sets of characters, Julia defines a **parametric type** called `Set{T}` whose parameter T can be any type. Parameters can have restrictions as well. Rational numbers, for example, can have `UInt8` or `Int64` components but not string components, so Julia provides the type `Rational{T<:Integer}`, restricting T to a descendant type of `Integer`. You'll notice several families of parametric types in Julia, including those for sets, arrays, ranges, and complex numbers.

Julia's array type is parameterized not only by the type of its elements but by its **dimension**. So the type of one-dimensional integer arrays differs from the type of two-dimensional integer arrays. Here are some examples of arrays:

$$a = \begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \end{bmatrix} \quad b = \begin{bmatrix} 5 & 3 & 7 \end{bmatrix} \quad c = \begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & 6 \end{bmatrix}$$

Here a is written in Julia as `[10,20,30,40]` and is a *one*-dimensional array, or **vector**, of type `Array{Int64,1}`. Array b is written `[5 3 7]` (note spaces instead of commas), and is a 1×3 (one row, three columns) *two*-dimensional array, or **matrix**, of type `Array{Int64,2}`. Array c is a 2×3 array, also with type `Array{Int64,2}`, and is written `[1 0 9; 0 1 6]`. Higher dimensional arrays have no special syntax, though they can be constructed with comprehensions, e.g., `[i+j+k for i=1:4, j=1:3, k=1:5]`.

In an n-dimensional array, the expression $a[e_1, e_2, ...e_n]$ will select a single element from the array when each of the indexes e_i are integers. However, the index expressions can also be ranges or vectors, allowing some very interesting **slices** to be computed:

```
a = [11 12 13 14; 21 22 23 24; 31 32 33 34]

@assert a[2,3] == 23                    # element in row 2, col 3
@assert a[2,2:4] == [22,23,24]          # row 2, cols 2 through 4
@assert a[1:3,3] == [13; 23; 33]        # rows 1 through 3 of col 3
@assert a[3,1:end] == [31,32,33,34]     # row 3, all elements
@assert a[3,:] == [31,32,33,34]         # row 3, all elements
@assert a[2,[1;3;4]] == [21,23,24]      # row 2, cols 1, 3, 4
@assert a[[1;3],[1;4]] == [11 14; 31 34] # very disjointed subarray
@assert a[7] == 13                      # enumerates by columns
```

Julia has *hundreds* of built-in methods for processing arrays, including `length(A)` for the number of elements in A, `ndims(A)` for the number of dimensions, and `size(A)` for a tuple of its dimensions. Some construct new arrays:

```
@assert zeros(Int64, 3) == [0, 0, 0]
@assert zeros(Int64, 2, 2) == [0 0; 0 0]
@assert ones(Int64, 3, 2) == [1 1; 1 1; 1 1]
@assert eye(Int32, 3) == [1 0 0; 0 1 0; 0 0 1]
@assert fill(5, 1, 4) == [5 5 5 5]
@assert transpose([1 3; 2 4]) == [1 2; 3 4]
@assert [1 3; 2 4]' == [1 2; 3 4]           # postfix ' transposes
@assert fill(10, 3, 1) == [10 10 10]'       # 2d column array!
@assert [2x for x in 1:5] == [2,4,6,8,10]
```

Many arithmetic operations extend to arrays. Array addition and subtraction (and a few other operations) are defined element-by-element, but multiplication, division, and exponentiation (and a few others) are not. Where an operation does not work element-by-element, a "dotted" version of the operator does work that way:

```
a = [1 2 3; 4 5 6; 7 8 9]
b = [1 0 0; 0 2 0; 9 9 9]

@assert a + b == [2 2 3; 4 7 6; 16 17 18]
@assert a * 2 == [2 4 6; 8 10 12; 14 16 18]
@assert a * b == [28 31 27; 58 64 54; 88 97 81]  # Matrix multiply
@assert a .* b == [1 0 0; 0 10 0; 63 72 81]      # Elementwise mul

@assert exp2(a) == [2.0 4.0 8.0; 16.0 32.0 64.0; 128.0 256.0 512.0]
```

Julia has a vast number of *built-in* functions useful in the field of linear algebra, including cross product; eigenvalue computation; Hessenberg and singular value decompositions; Givens rotations; computations of transpositions, determinants, triangles and diagonalizations; and a large number of factorizations. Dozens more methods are included in the standard library packages `Base.LinAlg.BLAS` and `Base.LinAlg.LAPACK`.

6.3.3 Sum and Product Types

Julia allows us to combine types T_1 and T_2 in an "algebraic" fashion to produce two new types, $T_1 + T_2$ and $T_1 \times T_2$. The **sum type** contains all the values from both types, while the **product type** contains pairs whose first element is from T_1 and the second from T_2. Let's build up these new types from `UInt8`s and `Bool`s:

	Sum	**Product**
Julia Type	`Union{UInt8, Bool}`	`Tuple{UInt8, Bool}`
Values	`0`, `1`, ... , `254`, `255`, `false`, `true`	`(0,false)`, `(0,true)`, `(1,false)`, `(1,true)`, ... `(255,false)`, `(255,true)`

Here are union types in action:

```
u = Union{UInt8, Bool}          # A new type
@assert typeof(u) == Union
@assert isa(u, Type)            # A union type is a type

@assert isa(0x08, u)            # UInt8 values belong
@assert isa(false, u)           # Boolean values belong
@assert isa(true, u)
@assert !isa(256, u)            # Not a UInt8, does not belong
```

and tuples:

```
t = Tuple{UInt8, Bool}          # a tuple type
@assert typeof(t) == DataType   # tuple types aren't special
@assert isa(t, Type)            # tuple types are types

@assert isa((0x08, false), t)   # tuples of the right type belong
@assert isa((0x7A, true), t)
@assert !isa((3, "wrong"), t)   # tuples of the wrong type do not
@assert !isa(0x25, t)           # non-tuples do not belong
```

You can remember the difference between sum types and product types by noting the following. There are 256 values in `UInt8` and 2 in `Bool`. The union (sum) type has $256+2 = 258$ values while the tuple (product) type has $256 \times 2 = 512$.

Sums and products are not limited to two constituent types:

```
# 3 component types
@assert typeof((4, false, [])) == Tuple{Int64, Bool, Array{Any, 1}}
@assert isa(false, Union{Int64, Bool, Array{Any, 1}})

# 1 component type
@assert typeof((5,)) == Tuple{Int64}
@assert isa(false, Union{Bool})
@assert Bool != Tuple{Bool}           # Because false != (false,)
@assert Bool == Union{Bool}           # Do you see why?

# No component types
@assert typeof(()) == Tuple{}
```

The type `Union{}` has no values (there are no underlying values to collect) while `Tuple{}` has exactly one, namely `()`, the zero-element tuple. If you've encountered abstract algebra, you'll recognize the analogy to 0 being the identity element of sum and 1 of product.

6.3.4 Type Annotations

Julia provides the ability to attach a type annotation to a variable, using the notation $v::T$ for variable v and type T in local variable declarations and assignments. All assignments of a value to a variable with a type annotation will be type-converted to the annotated type if possible. (In Julia, a value v is converted to type T by calling `convert(T,v)`.)

```
(function()
  local x::Int64 = 0x02
  local y = 0x02
  @assert typeof(x) == Int64  # The UInt8 was converted
  @assert typeof(y) == UInt8  # No annotations, no coversion

  # There's no conversion from String to Int64
  @assert isa(try x = "Oh no" catch (e) e end, Exception)
end)()
```

Type annotations may increase performance: a compiler can use knowledge that a variable has a given type to create efficient memory layout and run-time access code. Type annota-

tions are completely optional in Julia. Some languages (e.g. CoffeeScript) have no notion of type annotations, while some languages (e.g., Java, coming up in the next chapter) require them for all variables.

6.3.5 Covariance, Contravariance, and Invariance

We've seen subtype-supertype relationships among simple, nonparameterized types, for example `Int64 <: Number` (i.e., a 64-bit integer is a number). But what about parameterized, union, and tuple types? Can we say anything about their subtype-supertype status based on the types of their component types? For example, how are `Set{Int64}` and `Set{Number}` related, if at all? First, some terminology:

- If `Set{Int64} <: Set{Number}`, sets would be **covariant**
- If `Set{Number} <: Set{Int64}`, sets would be **contravariant**
- If neither holds, sets would be **invariant**
- If both hold, sets would be **bivariant**

It turns out that in Julia, sets are invariant. Why are they not covariant? If they were, we would be able to declare a variable a with declared type "set of animals" and assign to it a value of type "set of dogs." Since a is declared as a set of animals, it would *appear* reasonable to add a cat to the set. But the underlying value of the variable is a set of dogs, which cannot hold a cat. Invariance avoids this pitfall.

Tuples, on the other hand, *are* covariant:

```
t = (3, "hello")
@assert typeof(t) == Tuple{Int64, String}
@assert isa(t, Tuple{Real, String})
@assert isa(t, Tuple{Integer, AbstractString})
@assert isa(t, Tuple{Number, Any})

@assert Tuple{Symbol, Float16, Union{}} <: Tuple{Any, Real, Any}
```

The previous problem with sets does not arise because tuples are immutable.

6.4 MULTIPLE DISPATCH

Let's turn now to functions and consider the following problem: how can we write a function whose behavior depends on the *types* of its arguments? For example, consider a function called `times` such that:

- If x and y are numbers, the function multiplies, e.g., `times(8,3)` ⇒ `24`
- If x is a string and y is an unsigned integer, we have repetition, e.g., `times("ho",3)` ⇒ `"hohoho"`
- If x is a vector and y is a number, we have element-wise multiplication, e.g., `times([1,2,3],3)` ⇒ `[3,6,9]`
- If x is a number and y is a vector, we also have element-wise multiplication, e.g., `times(5,[1,2,3])` ⇒ `[5,10,15]`

will wait for the function to complete on the remote process, and produce the function's result. The following script creates three new processes,[4] and instructs process #4 to factor a large number:[5]

```
addprocs(3)
using Primes.factor
@assert nprocs() == 4
future = remotecall(factor, 4, 21883298135690819)
@assert isa(future, Future)
factors = fetch(future)
@assert factors == Dict(234711901=>1,93234719=>1)
```

The `@spawn` macro is often used in place of `remotecall`; it not only takes advantage of a clearer syntax (since macros take unevaluated expressions as arguments), but it chooses a process for you:

```
addprocs(3)
using Primes.factor
ref = @spawn factor(21883298135690819)
factors = fetch(ref)
@assert factors == Dict(234711901=>1,93234719=>1)
```

Between the remote call and fetching the result, the calling process is free to do other work.

A common example of parallel programming across multiple cores computes an approximation of π by generating random points in the square $(0,0)...(1,1)$. Counting the number of random points within the inscribed arc (the points (x, y) for which $x^2+y^2 \leq 1$ in Figure 6.3) divided by the total number of points within the square approximates the value $\frac{\pi}{4}$. Let's generate a million random points in parallel on three cores:

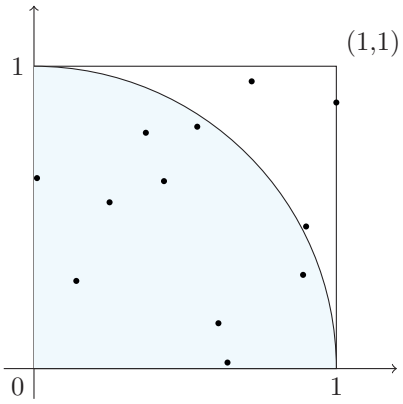

Figure 6.3 Approximation of $\frac{\pi}{4}$

[4]Calling `addprocs` with an integer argument creates processors on the local machine to take advantage of multiple cores. There are other variants of `addprocs` to add processors on remote machines either via user-host-port strings or cluster managers; we will not cover these alternatives in this text.

[5]The `factor` function resides in the package `Primes`. You must first install this package by executing `Pkg.add("Primes")` in an interactive shell, or `julia -e 'Pkg.add("Primes")'` from the command line.